Modern C++ Programming

8. Basic Concepts VI

Functions and Preprocessing

Federico Busato

2026-01-06

Table of Contents

Table of Contents

3 Lambda Expressions

4 Preprocessing

Functions

Overview

A **function** (**procedure** or **routine**) is a piece of code that performs a *specific task*

Purpose:

- **Avoiding code duplication**: less code for the same functionality $\rightarrow$ less bugs
- **Readability**: better express what the code does
- **Organization**: break the code in separate modules

Function Parameter and Argument

Function Parameter [formal]

A **parameter** is the variable which is part of the method signature

Function Argument [actual]

An **argument** is the actual value (instance) of the variable that gets passed to the function

```
void f(int a, char* b); // parameters: int a, char* b
                        // return type: void

f(3, "abc");            // arguments: 3, "abc"
```

Pass by-Value

Call-by-value

The object is copied and assigned to input arguments of the method `f(T x)`

Advantages:

- Changes made to the parameter inside the function have no effect on the argument

Disadvantages:

- Performance penalty if the copied arguments are large (e.g. a structure with several data members)

When to use:

- Built-in data type and small objects ($\leq$ 8 bytes)

When not to use:

- Fixed size arrays which decay into pointers
- Large objects

Pass by-Pointer

Call-by-pointer

The address of a variable is copied and assigned to input arguments of the method

`f(T* x)`

Advantages:

- Allows a function to change the value of the argument
- The argument is not copied (fast)

Disadvantages:

- The argument may be a null pointer
- Dereferencing a pointer is slower than accessing a value directly

When to use:

- *Raw* arrays (use `const T*` if read-only)

When not to use:

- All other cases

Pass by-Reference

Call-by-reference

The reference of a variable is copied and assigned to input arguments of the method `f(T& x)`

Advantages:

- Allows a function to change the value of the argument (better readability compared with pointers)
- The argument is not copied (fast)
- References must be initialized (no null pointer)
- Avoid implicit conversion (without `const T&`)

When to use:

- All cases except raw pointers

When not to use:

- Pass by-value *could* give performance advantages and improve the readability with built-in data type and small objects that are trivially copyable

Examples

```cpp
struct MyStruct;

void f1(int a);          // pass by-value
void f2(int& a);         // pass by-reference
void f3(const int& a);   // pass by-const reference
void f4(MyStruct& a);    // pass by-reference

void f5(int* a);         // pass by-pointer
void f6(const int* a);   // pass by-const pointer
void f7(MyStruct* a);    // pass by-pointer

void f8(int*& a);        // pass a pointer by-reference
//--------------------------------------------------------------
char c = 'a';
f1(c);      // ok, pass by-value (implicit conversion)
// f2(c); // compile error different types
f3(c);      // ok, pass by-value (implicit conversion)
```

Signature

Function signature defines the *input types* for a (specialized) function and the *inputs + outputs types* for a template function

A function signature includes the <u>number</u> of arguments, the <u>types</u> of arguments, and the <u>order</u> of the arguments

- The C++ standard prohibits a function declaration that only differs in the return type
- Function declarations with different signatures can have distinct return types

Overloading

Function overloading allows having distinct functions with the same name but with different *signatures*

```cpp
void f(int a, char* b);           // signature:  (int, char*)

// char f(int a, char* b);        // compile error same signature
                                  // but different return types

void f(const int a, char* b);     // same signature, ok
                                  // const int == int

void f(int a, const char* b);     // overloading with signature: (int, const char*)

int f(float);                     // overloading with signature:  (float)
                                  // the return type is different
```

GCC 14 adds the flag `-fdiagnostics-all-candidates` to show all function candidates when overload resolution failure occurs

Overloading Resolution Rules

- An exact match
- A promotion (e.g. `char` to `int`)
- A standard type conversion (e.g. `float` and `int`)
- A constructor or user-defined type conversion ⇝

```
void f(int   a);
void f(float b);          // overload
void f(float b, char c); // overload
//------------------------------------------------------------
   f(0);         // exact match
   f('a');       // promotion from char to int (promotion)
// f(3LL);       // compile error ambiguous match
   f(2.3f);      // exact match
// f(2.3);       // compile error ambiguous match
   f(2.3, 'a'); // standard type conversion, ambiguity is not possible here
```

Overloading and =delete

`=delete` can be used to prevent calling the wrong overload

```
void g(int) {}

void g(double) = delete;

g(3);    // ok
#include <cstddef> // std::nullptr_t

void f(int*) {}

void f(std::nullptr_t) = delete;

f(nullptr); // compile error
```

Function Default Parameters

Default/Optional parameter

A **default parameter** is a function parameter that has a default value

- If the user does not supply a value for this parameter, the default value will be used
- All default parameters must be the rightmost parameters
- Default parameters must be declared only once
- Default parameters can improve compile time and avoid redundant code because they avoid defining other overloaded functions

```cpp
void f(int a, int b = 20);          // declaration

//void f(int a, int b = 10) { ... } // compile error, already set in the declaration

void f(int a, int b) { ... }        // definition, default value of "b" is already set

f(5);  // b is 20
```

Function Pointers and Function Objects

Standard C achieves generic programming capabilities and composability through the concept of **function pointer**

A function can be passed as a pointer to another function and behaves as an *"indirect call"*

```
#include <stdlib.h> // qsort

int descending(const void* a, const void* b) {
    return *((const int*) a) > *((const int*) b);
}

int array[] = {7, 2, 5, 1};
qsort(array, 4, sizeof(int), descending);
// array: { 7, 5, 2, 1 }
```

```cpp
int eval(int a, int b, int (*f)(int, int)) {
    return f(a, b);
}
// type: int (*)(int, int)
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

cout << eval(4, 3, add); // print 7
cout << eval(4, 3, sub); // print 1
```

Problems:

Safety There is no check of the argument type in the generic case (e.g. `qsort`)

Performance Any operation requires an indirect call to the original function. Function inlining is not possible

Function Object

A **function object**, or **functor**, is a *callable* object that can be treated as a parameter

C++ provides a more efficient and convenient way to pass *"procedure"* to other functions called **function object**

```
#include <algorithm> // for std::sort

struct Descending {                    // <-- function object
    bool operator()(int a, int b) {  // function call operator
        return a > b;
    }
};
int array[] = {7, 2, 5, 1};
std::sort(array, array + 4, Descending{});
// array: { 7, 5, 2, 1 }
```

Advantages:

Safety Argument type checking is always possible. It could involve templates

Performance The compiler injects `operator()` in the code of the destination function and then compile the routine. Operator inlining is the standard behavior

C++11 simplifies the concept by providing less verbose `function objects` called **lambda expressions**

Lambda Expressions

Lambda Expression

A C++11 **lambda expression** is an *inline local-scope* function object

```
auto x = [capture clause] (parameters) { body }
```

The expression to the right of `=` is the **lambda expression**.

The runtime object `x` created by that expression is the **closure**

```cpp
auto descending = [](int a, int b) { return a > b; };

// equivalent to (simplified)
struct Descending {
    bool operator()(int a, int b) { return a > b; }
};
Descending descending;
```

Lambda Expression

```
auto x = [capture clause] -> <type> { body }
```

`[capture clause]` defines how the local scope arguments are captured (by-value, by-reference, etc.)

`parameters` are normal function parameters (optional in C++23*)

`body` is a normal function body (*function call operator*)

`-> <type>` trailing return type (optional)

Additionally, *lambda expressions* support *template and concepts* in C++20 and *function attributes* in C++23

* some compilers support lambda expressions without parameters in previous C++ standards

Lambda Expression Examples

```cpp
#include <algorithm> // for std::sort

int array[] = {7, 2, 5, 1};
auto lambda = [](int a, int b){ return a > b; }; // named lambda

std::sort(array, array + 4, lambda);
// array: { 7, 5, 2, 1 }

// in alternative, in one line of code:          // unnamed lambda
std::sort(array, array + 4, [](int a, int b){ return a > b; });
// array: { 7, 5, 2, 1 }

auto lambda2 = []{ return 3; };           // no parameters, C++23

auto lambda3 = [] static { return 3; }; // static function call operator, C++23
```

Capture List

Lambda expressions *capture* external variables used in the body of the lambda in two ways:

- Capture *by-value*
- Capture *by-reference* (can modify external variable values)

Capture list can be passed as follows

- `[]` no capture
- `[=]` captures all variables *by-value*
- `[&]` captures all variables *by-reference*
- `[var1]` captures only `var1` *by-value*
- `[&var2]` captures only `var2` *by-reference*
- `[var1, &var2]` captures `var1` *by-value* and `var2` *by-reference*

Capture List Examples

```
// GOAL: find the first element greater than "limit"
#include <algorithm> // for std::find_if
int limit = ...

auto lambda1 = [=](int value)      { return value > limit; }; // by-value
auto lambda2 = [&](int value)      { return value > limit; }; // by-reference
auto lambda3 = [limit](int value)  { return value > limit; }; // "limit" by-value
auto lambda4 = [&limit](int value) { return value > limit; }; // "limit" by-reference
// auto lambda5 = [](int value)    { return value > limit; }; // no capture
                                                              // compile error
int array[] = {7, 2, 5, 1};
std::find_if(array, array + 4, lambda1);
```

Capture List - Other Cases

- `[=, &var1]` captures all variables used in the body of the lambda **by-value**, except `var1` that is captured **by-reference**
- `[&, var1]` captures all variables used in the body of the lambda **by-reference**, except `var1` that is captured **by-value**
- `[new_var = var1]`, `[&new_var = var1]` introduce a new value or reference `new_var` initialized by `var1` C++14
- A lambda expression can read a variable without capturing it if the variable is `constexpr`

```
constexpr int limit = 5;
int var1 = 3, var2 = 4;

auto lambda1 = [](int value){ return value > limit; };

auto lambda2 = [=, &var2]() { return var1 > var2;   };
```

Lambda Behind the Hood

The following code

```
int   a;
float b;
auto  lambda = [a, &b](int v) {return 4;};
```

is roughly equivalent to

```
struct /*unnamed*/ {
   int    a; // private
   float& b; // private

   inline /*constexpr*/ int operator()(int v) const {
       return 4;
   }
} lambda;
```

Lambda Expression and Function Relation

A *lambda expression* can be converted to a function (*stateless*) if its capture list is empty

```
// lambda_func is equivalent to
// int lambda_func(int first, int second){ return first + second; };

void f(int (lambda_func)(int, int)) {
    cout << lambda_func(2, 3);
}

auto lambda = [](int first, int second){ return first + second; };
f(lambda); // print 5
```

Parameter Notes

C++14 Lambda expression parameters can be automatically deduced

```
auto x = [](auto value) { return value + 4; };
```

C++14 Lambda expression parameters can be initialized

```
auto x = [](int i = 6) { return i + 4; };
```

Lambda expressions can be composed

```
auto lambda1 = [](int value){ return value + 4; };
auto lambda2 = [](int value){ return value * 2; };

auto lambda3 = [&](int value){ return lambda2(lambda1(value)); };
// returns (value + 4) * 2
```

A function can return a lambda

(dynamic dispatch is also possible if the capture list is empty)

```
auto f() {
    return [](int value){ return value + 4; };
}
auto lambda = f();
cout << lambda(2); // print "6"
```

A lambda expression can contain another lambda expression

```cpp
auto lambda1 = [](auto value) {
    int  x       = 5;
    auto lambda2 = [=](auto v) { return x * value + v; };
    return lambda2(3);
};
cout << lambda1(2); // print "13"
```

Recursion ★

Lambda expressions can be called recursively

```
auto factorial = [](int n, auto fac) {
    return (n <= 1) ? 1 : n * fac(n - 1, fac);
};
factorial(5, factorial);
```

C++23 allows to access the `this` pointer of a lambda object with the syntax `this auto` as first parameter

```
auto factorial = [](this auto self, int n) -> int {  // or 'this auto&&'
    return (n <= 1) ? 1 : n * self(n - 1);
};
factiorial(5);
```

`constexpr`/`consteval` Lambda Expression

C++17 Lambda expressions are implicitly `constexpr` (if they satisfy the requirements of a `constexpr` function). Lambda expressions can be also explicitly marked `constexpr`

C++20 Lambda expressions support `consteval`

```
auto factorial = [](int value) constexpr {
    int ret = 1;
    for (int i = 2; i <= value; i++)
        ret *= i;
    return ret;
};
auto mul = [](int v) consteval { return v * 2; };
auto add = [](int x) { return x + 3; };

constexpr int v1 = factorial(4) + mul(5) + add(3); // '24' + '10' + '5'
```

C++20 Lambda expression supports `template` and `requires` clause

```
auto lambda = []<typename T>(T value)
              requires std::is_arithmetic_v<T> {
    return value * 2;
};
auto v = lambda(3.4); // v: 6.8 (double)
// lambda(nullptr);   // compiler error
```

Before C++20, `template` arguments can be emulated with `auto` + `decltype`

```
auto lambda = [](auto value) {
    using T = decltyle(value);  // T: double
};
lambda(3.4);
```

Lambda and template without automatic deduction needs the full syntax

```
auto lambda = []<typename T>(int value) {
    return value * sizeof(T);
};

// lambda<double>(3);          // compiler error
lambda.operator()<double>(3); // ok
```

mutable Lambda Expression ★

Lambda capture is *by-const-value*

mutable specifier allows the lambda to modify the parameters captured *by-value*

```
int  var = 1;

auto lambda1 = [&](){ var = 4; };          // ok
lambda1();
cout << var; // print '4'

// auto lambda2 = [=](){ var = 3; };       // compile error
// lambda operator() is const

auto lambda3 = [=]() mutable { var = 3; }; // ok
lambda3();
cout << var;  // print '4', lambda3 captures by-value
```

Capture List and Classes ⇝

- `[this]` captures the current object `(*this)` *by-reference* (implicit in C++17)
- `[=]` default capture of `this` pointer by value has been deprecated C++20
- `[new_var = x]`, `[&new_var = x]` introduce a new value or reference `new_var` initialized by `x` C++14

```
class A {
    int data = 1;

    void f() {
        int  var     = 2;                            // <-- local variable
        auto lambda1 = [=]()     { return var;  };        // copy by-value, return 2
        auto lambda2 = [=]()     { int var = 3; return var; }; // return 3 (nearest scope)
        auto lambda3 = [this]()  { return data; };        // copy by-reference, return 1
        auto lambda4 = [*this]() { return data; };        // copy by-value (C++17), return 1
//      auto lambda5 = [data]()  { return data; };        // compile error 'data' is not visible
        auto lambda6 = [y = data]() { return y; };        // return 1
    }
};
```

Preprocessing

Preprocessing and Macro

A **preprocessor directive** is any line preceded by a *hash* symbol (#) which tells the compiler how to interpret the source code *before* compiling it

Macro are preprocessor directives which substitute any occurrence of an *identifier* in the rest of the code by replacement

Macro are evil:

Do not use macro expansion!!

...or use as little as possible

- Macro cannot be directly debugged
- Macro expansions can have unexpected side effects
- Macro have no namespace or scope

All statements starting with `#`

- `#include "my_file.h"`
  Inject the code in the current file
- `#define MACRO <expression>`
  Define a new macro
- `#undef MACRO`
  Undefine a macro
  (a macro should be undefined as early as possible for safety reasons)

Multi-line Preprocessing: `\` at the end of the line

Indent: `#   define`

Conditional Compiling

- ```
  #if <condition 1>
      ..code..
  #elif <condition 2>
      ..code..
  #else
      ..code..
  #endif
  ```
- Check if a macro is defined
  ```
  #if defined(MACRO)   // equal to #ifdef  MACRO
  #elif defined(MACRO) // equal to #elifdef MACRO in C++23
  ```
- Check if a macro is NOT defined
  ```
  #if !defined(MACRO)   // equal to #ifndef  MACRO
  #elif !defined(MACRO) // equal to #elifdef MACRO in C++23
  ```
  ```

Common Error 1 - Assuming macros have file scope

Define macros in header files and before includes!!

```cpp
#include <iostream>
#define value           // <- very dangerous!!
#include "big_lib.hpp"

int main() {
    std::cout << add_3(4); // should print 7, but it always prints 3
}
```

big_lib.hpp:

```cpp
int add_3(int value) {    // 'value' disappears
    return value + 3;
}
```

It is very hard to see this problem when the macro is in a header

`#if defined` **can introduce bugs related to macro visibility**

```cpp
#include "header1.hpp"
#include "header2.hpp"
// ... many other headers and/or big project ...

#if defined(ENABLE_DEBUG)  // is ENABLE_DEBUG defined here?
    int f(int v) { cout << v << endl; return v * 3; } // first path
#else
    int f(int v) { return v * 3; }                    // second path
#endif
```

Fixing the problem...the wrong way:

```
#if ENABLE_DEBUG // or #if ENABLE_DEBUG == 1
    void f(int v) { cout << v << endl; return v * 3; } // first path
...
```

The second path is enabled when `ENABLE_DEBUG` defined to `0` and when the macro is not defined, *potentially not intentionally*.

`ENABLE_DEBUG` is evaluated as `0` if it is NOT defined.

Furthermore, even the most common warning flags (`-Wall -Wextra -Wpedantic`) don't raise the issue. The user needs to explicitly add `-Wundef` to detect the problem

Solution: **Function-like macros**

```
#define ENABLE_DEBUG() 1
...
#if ENABLE_DEBUG() // compile error if it is not defined
```

Common Error 3 - Parenthesis

Forget to use parenthesis in macro definitions!!

```
#define SUB1(a, b)  a - b       // WRONG
#define SUB2(a, b) (a - b)      // WRONG
#define SUB3(a, b) ((a) - (b))  // correct

cout << (5 * SUB1(2, 1));   // print 9 not 5!!
cout << SUB2(3 + 3, 2 + 2); // print 6 not 2!!
cout << SUB3(3 + 3, 2 + 2); // print 2
```

Common Error 4 - Assuming macros are like common code to debug

Macros make hard to find compile errors!!

```
 1: #include <iostream>
 2:
 3: #define F(a) {        \
 4:         ...          \
 5:         ...          \
 6:         return v;
 7:
 8: int main() {
 9:     F(3);      // compile error at line 9!!
10: }
```

- **In which line is the error??!***

*modern compilers are able to roll out the macro with `-g3` flag

Common Error 5 - Arguments evaluation

Macro can introduce bugs related to the evaluation of their expressions!!

```
#if defined(DEBUG)
#   define CHECK(EXPR)   // do something with EXPR
    void check(bool b) { /* do something with b */ }
#else
#   define CHECK(EXPR)   // do nothing
    void check(bool) {}  // do nothing
#endif
bool clear_system_error() { /* change program state;
                               return true if everything is fine */ }
check( clear_system_error() )
CHECK( clear_system_error() ) // <-- problem here
```

- **What happens when `DEBUG` is not defined?**

  `f()` is not evaluated by using the macro

Common Error 6 - Multi-line macros

Forget curly brackets in multi-lines macros!!

```cpp
#include <iostream>
#include <nuclear_explosion.hpp>

#define NUCLEAR_EXPLOSION                   \  // {
    std::cout << "start nuclear explosion"; \
    nuclear_explosion();
                                               // }

int main() {
    bool never_happen = false;
    if (never_happen)
        NUCLEAR_EXPLOSION
} // BOOM!! ☠
```

The second line is executed!!

Common Error 7 - Assuming macros have local scope

Macros do not have scope!!

```cpp
#include <iostream>

void f() {
    #define value 4
    std::cout << value;
}
int main() {
    f();                // 4
    std::cout << value; // 4
    #define value 3
    f();                // 4
    std::cout << value; // 3
}
```


* In general, compilers raise a warning for multiple definitions of the same macro

Common Error 8 - Assuming macros behave like functions

Macros can have side effect!!

```c
#define MIN(a, b) ((a) < (b) ? (a) : (b))

int main() {
    int array1[] = { 1, 5, 2 };
    int array2[] = { 6, 3, 4 };
    int i        = 0;
    int j        = 0;
    int v1       = MIN(array1[i++], array2[j++]); // v1 = 5!!
    int v2       = MIN(array1[i++], array2[j++]); // undefined behavior/
}                                                 // segmentation fault ☠
```

arne-mertz.de/2019/03/macro-evil

Common Error 9 - Undefined behavior

Macros can have undefined behavior themselves!!

```c
#define MY_MACRO defined(EXTERNAL_MACRO)

#if MY_MACRO
#   define MY_VALUE 1
#else
#   define MY_VALUE 0
#endif

int x = MY_VALUE; // undefined behavior: 'defined' has a different meaning
                  // if outside a conditional preprocessioning directive, e.g. #if
```

When Preprocessors are Necessary

- **Conditional compiling**: different architectures, compiler features, etc.
- **Mixing different languages**: code generation (example: `asm assembly`)
- **Complex name replacing**: see template programming

Otherwise, prefer `const` and `constexpr` for constant values and functions

```
#define SIZE 3      // replaced with
const int SIZE = 3; // only C++11 at global scope

#define SUB(a, b) ((a) - (b)) // replaced with
constexpr int sub(int a, int b) {
    return a - b;
}
```

`__LINE__`	Integer value representing the current line in the source code file being compiled
`__FILE__`	A string literal containing the name of the source file being compiled
`__FUNCTION__`	(non-standard, `gcc`, `clang`) A string literal containing the name of the function in the 'macro scope'
`__PRETTY_FUNCTION__`	(non-standard, `gcc`, `clang`) A string literal containing the full signature of the function in the 'macro scope'
`__func__`	(C++11 keyword) A string containing the name of the function in the 'macro scope'

source.cpp:

```cpp
#include <iostream>

void f(int p) {
    std::cout << __FILE__ << ":" << __LINE__; // print 'source.cpp:4'
    std::cout << __FUNCTION__;                // print 'f'
    std::cout << __func__;                    // print 'f'
}

// see template lectures
template<typename T>
float g(T p) {
    std::cout << __PRETTY_FUNCTION__;         // print 'float g(T) [T = int]'
    return 0.0f;
}

void g1() { g(3); }
```

C++20 provides source location utilities for replacing macro-based approach

`#include <source_location>`

`current()`	get source location info (`static member`)
`line()`	source code line
`column()`	line column
`file_name()`	current file name
`function_name()`	current function name

```
#include <source_location>

void f(std::source_location s = std::source_location::current()) {
    cout << "function: " << s.function_name() << ", line " << s.line();
}
f(); // print: "function: f, line 6"
```

Select code depending on the C/C++ version

- `#if defined(__cplusplus)` C++ code
- `#if __cplusplus == 201103L` ISO C++ 2011*
- `#if __cplusplus == 201402L` ISO C++ 2014*
- `#if __cplusplus == 201703L` ISO C++ 2017

Select code depending on the compiler

- `#if defined(__GNUG__)` The compiler is gcc/g++ †
- `#if defined(__clang__)` The compiler is `clang/clang++`
- `#if defined(_MSC_VER)` The compiler is `Microsoft Visual C++`

---

* `MSVC defines __cplusplus == 199711L even for C++11/14`

† `__GNUC__ is defined by many compilers, e.g clang`

Condition Compiling Macros 2/2

Select code depending on the operating system or environment

- `#if defined(_WIN64)` OS is `Windows 64-bit`
- `#if defined(__linux__)` OS is `Linux`
- `#if defined(__APPLE__)` OS is `Mac OS`
- ...and many others

`__DATE__` A string literal in the form "MMM DD YYYY" containing the date in which the compilation process began

`__TIME__` A string literal in the form "hh:mm:ss" containing the time at which the compilation process began

Other Macros

Very comprehensive macro list:

- `Pre-defined Compiler Macros wiki`
- `Boost.Predef`
- `How to detect the operating system type using compiler predefined macros`

Feature Testing Macro

C++17 introduces `__has_include` macro which returns `1` if header or source file with the specified name exists

```
#if __has_include(<iostream>)
#   include <iostream>
#endif
```

C++20 introduces a set of macros to evaluate if a given feature is supported by the compiler

```
#if __cpp_constexpr
constexpr int square(int x) { return x * x; }
#endif
```

Common Error 10 ⇝

Macros depend on compilers and environment!!

```cpp
struct A {
    int x;  // enable C++11 code
#if __cplusplus >= 201103
    A() = default;
#else
    A() {}
#endif
};
```

```cpp
// should return ≈ 10.0f
float safe_function() {
    A a{}; // zero-initialization
    for (int i = 0; i < 10; i++)
        a.x += 1.0f;
    return a.x;
}
// what is the behavior ???
```

The code works fine on Linux, but not under Windows MSVC. MSVC sets `__cplusplus` to `199711` even if C++11/14/17 flag is set!! in this case the code can return `NaN`

see Lecture "Object-Oriented Programming II - Zero Initialization" and MSVC now correctly reports `__cplusplus`

Stringizing Operator (#)

The **stringizing macro operator** (`#`) causes the corresponding actual argument to be enclosed in double quotation marks `"`

```
#define STRING_MACRO(string) #string

cout << STRING_MACRO(hello);  // equivalent to "hello"
```

```
#define INFO_MACRO(my_func)                          \
{                                                    \
    my_func                                          \
    cout << "call " << #my_func << " at "            \
         << __FILE__ << ":" __LINE__;                \
}
void g(int) {}

INFO_MACRO( g(3) ) // print: "call g(3) at my_file.cpp:7"
```

Common Error 11

Code injection

```
#include <cstdio>

#define CHECK_ERROR(condition)                                  \
{                                                               \
    if (condition) {                                            \
        std::printf("expr: " #condition " failed at line %d\n",\
                    __LINE__);                                  \
    }                                                           \
}

int t = 6, s = 3;
CHECK_ERROR(t > s) // print "expr: t > s failed at line 13"
CHECK_ERROR(t % s == 0) // segmentation fault!!! ☠
// printf interprets "% s" as a format specifier
```

`#error` and `#warning`

- `#error "text"` The directive emits a user-specified error message at compile time when the compiler parse it and stop the compilation process

- C++23 `#warning "text"` The directive emits a user-specified warning message at compile time when the compiler parse it without stopping the compilation process

`#pragma`

The `#pragma` directive controls implementation-specific behavior of the compiler. In general, it is not portable

- `#pragma message "text"` Display informational messages at compile time (every time this instruction is parsed)
- `#pragma GCC diagnostic warning "-Wformat"`
  Disable a GCC warning
- `_Pragma(<command>)` (C++11)
  It is a keyword and can be embedded in a `#define`

```cpp
#define MY_MESSAGE \
    _Pragma("message(\"hello\")")
```

Token-Pasting Operator (##) ★

The **token-concatenation (or pasting) macro operator** (`##`) allows combining two tokens (without leaving no blank spaces)

```
#define FUNC_GEN_A(tokenA, tokenB)  \
    void tokenA##tokenB() {}

#define FUNC_GEN_B(tokenA, tokenB)  \
    void tokenA##_##tokenB() {}

FUNC_GEN_A(my, function)
FUNC_GEN_B(my, function)

myfunction();  // ok, from FUNC_GEN_A
my_function(); // ok, from FUNC_GEN_B
```

Variadic Macro ★

A **variadic macro** C++11 is a special macro accepting a variable number of arguments (separated by comma)

Each occurrence of the special identifier `__VA_ARGS__` in the macro replacement list is replaced by the passed arguments

Example:

```
void f(int a)                { printf("%d", a);          }
void f(int a, int b)         { printf("%d %d", a, b);    }
void f(int a, int b, int c)  { printf("%d %d %d", a, b, c); }

#define PRINT(...) \
    f(__VA_ARGS__);

PRINT(1, 2)
PRINT(1, 2, 3)
```

Macro Trick ⋆

Convert a number literal to a string literal

```
#define TO_LITERAL_AUX(x) #x
#define TO_LITERAL(x)     TO_LITERAL_AUX(x)
```

Motivation: avoid integer to string conversion (performance)

```
int main() {
   int  x1   = 3 * 10;
   int  y1   = __LINE__ + 4;
   char x2[] = TO_LITERAL(3);
   char y2[] = TO_LITERAL(__LINE__);
}
```